# Project2 :Barrier Synchronization

Jyoti Saini, Raj Kripal Danday

**Introduction:**

Barriers are a type of synchronization method used in multithreaded programming. A barrier for a group of threads or processes means any thread/process must stop at this point and other threads/process cannot proceed until all other threads reach this barrier. One application of the barrier is to make sure that all threads have updated values in shared data structure before any thread could use it further for its next iteration or step. Libraries Like OpenMP and MPI provide standards for parallel computing which eases the construction of such barriers. We have implemented barriers using both OpenMP and MPI. We also implemented an OpenMP-MPI barrier to synchronize multiple threads and processors.

**OpenMP Barriers :**

OpenMP barriers are used to synchronize multiple threads running on a same processor, to bring the threads to a certain standpoint. We implemented 2 barriers for OpenMp from MCs paper

1. **Dissemination Barrier** : Dissemination barrier is a round-based barrier and every thread has its own view of all the other threads involved in the synchronization. In Dissemination barrier each processor signals the processor (i+2^k)mod N in kth round. We need k = ceil(log N) rounds to synchronize all the processes. Every thread has flag for every round and every parity ( 0 or 1). Each processor has localFlags, a pointer to structure which holds its own flags as well as pointer to the partner's processor flag. Each processor spin on its local myFlags. When a processor reaches a barrier, it informs its partner by setting its flag through pointer partnerFlag.

2. **MCS Tree barrier (4-ary) :** The MCS Tree Barrier was implemented similar to the MCS paper. There is a 4-ary arrival tree and a binary wakeup tree. Initially the data structures are assigned for each process where the parent-child relations are established using pointers for both the trees. Each child informs its parent by using the parent pointer from the arrival tree. Once the last process enters the arrival tree the wakeup tree is triggered and the sense is reversed based on which the threads start exiting the barrier. Here the parent informs the child that it may exit. The nodes on which the threads spin are dynamically allocated in order to reduce false sharing.

**MPI Barriers :**

MPI barriers are used to synchronize different processors on an SMP system. We have implemented 2 barriers in MPI from MCS paper.

1. **Tournament Barrier** : In the tournament barrier, the barrier is achieved using a hierarchical tree structure in consecutive rounds. There are N number of threads and log2(N) rounds. At each round we statically define winner and loser for that round. Thus we know that winners will advance to the next level and losers will sleep or wait. Thus we have a pairing of two threads after each round.We keep two sense flag for each loser and winner. One flag for its own and other hold the address of the opponent flag. When the champion reached the topmost position, every thread would have reached a certain standpoint. The winner wakes up all the losers in consecutive rounds. This is done by the winner by blocking send/receive messages to the opponent. Since the winners and losers are statically determined , we know who is loser and who is winner in the particular round and this helps us achieve barrier synchronization. The contention for a global variable is eliminated and there is no polling for any memory hence there is no network traffic. The barrier is achieved from the combination of small barriers. The loser in the round tells the winner that it has reached the level and winner should move on to the next level. This is done by blocking send/receive messages from the opponent. Winners wait and receive until loser come about telling them their arrival. The losers spin on send/wait from the winner in the wake up tree. On the wake up tree traversal the winner just send to the corresponding opponent loser to break out the spinning phase. After all the threads are woken up, threads move to the next barrier.

2. **Dissemination Barrier:** The MPI dissemination barrier implemented uses pure message passing and has no shared data. After computing the number of rounds using the number of processors which are participating in the barrier, the MPI_Isend() and MPI_Irecv() non-blocking message passing functions are used to achieve the barrier. As the calls are non-blocking, an MPI_Wait() call waits for the received message by spinning on the status of the MPI_Irecv() call. The non-blocking calls are used instead of the blocking calls in order to avoid deadlocks due to circular waiting of both the send and receive calls.

**OpenMPI-MPI combined barrier :**

We combined Dissemination barrier from MPI and MCS tree barrier from OpenMP to get a combined barrier. We synchronize every thread in every process and then synchronize each processor. We did so as each processor has some amount of work to do.

**Experiments Conducted:**

In our experiments, we varied the load by giving each thread/processor a computation heavy task. The task here was to find the $n^{th}$ prime number. By varying n, we could vary the load on each thread/processor. In order to get a reasonable result we have averaged out the time spent by a thread/processor inside a barrier by looping over 100,000 barriers in case of pure openMP or MPI barriers and over 1,000 barriers in case of the mixed openMP-MPI barrier.

Note: As fourcore was not functional we had to run the OpenMP experiments on the six core. We have assigned one thread per processor and ran the experiment for 2, 4, 6 and 8 threads to gather our results.

**OpenMP :**

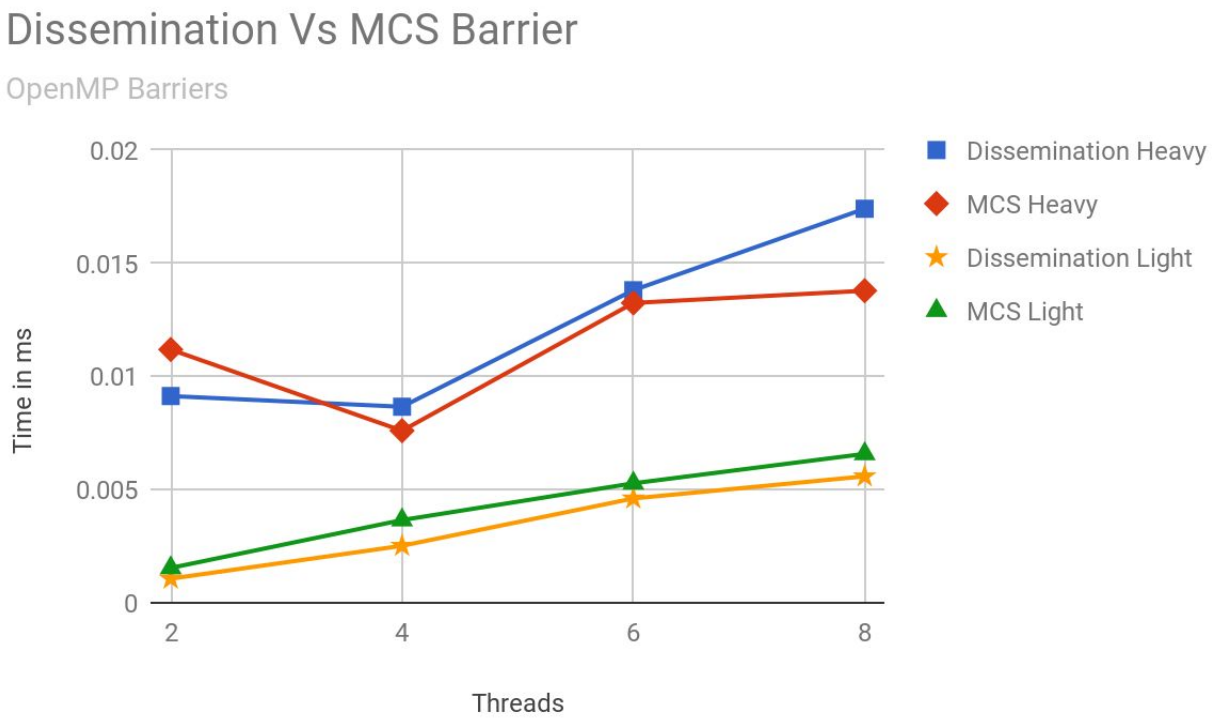


**Figure 1: Comparison of wait-times for heavily and lightly loaded threads for OpenMP Barriers.**

**MPI:**

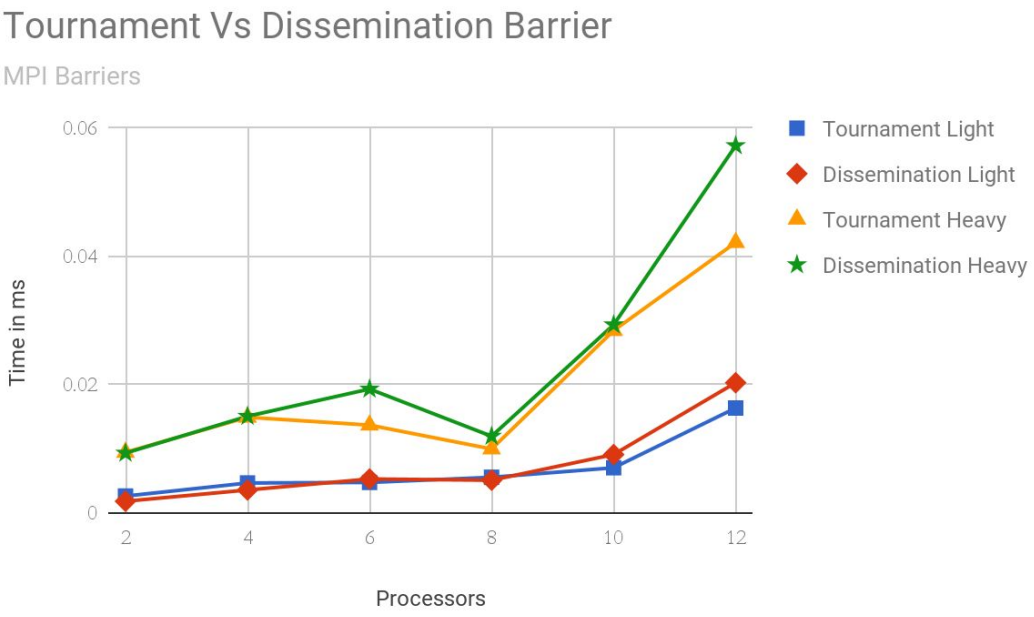


**Figure 2: Comparison of wait-times of heavily and lightly loaded processors for MPI Barriers.**

**OpenMP-MPI :**

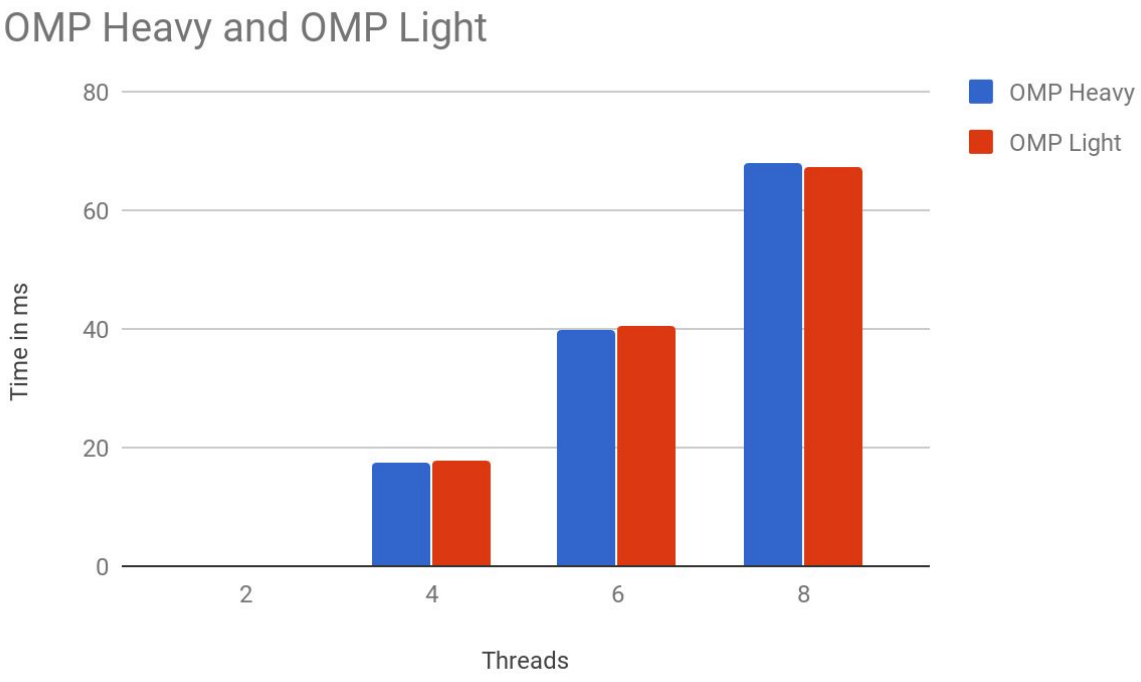


**Figure 3: Comparison of heavily and lightly loaded threads wait-time for the OpenMP barrier part of the OpenMP-MPI Barrier**

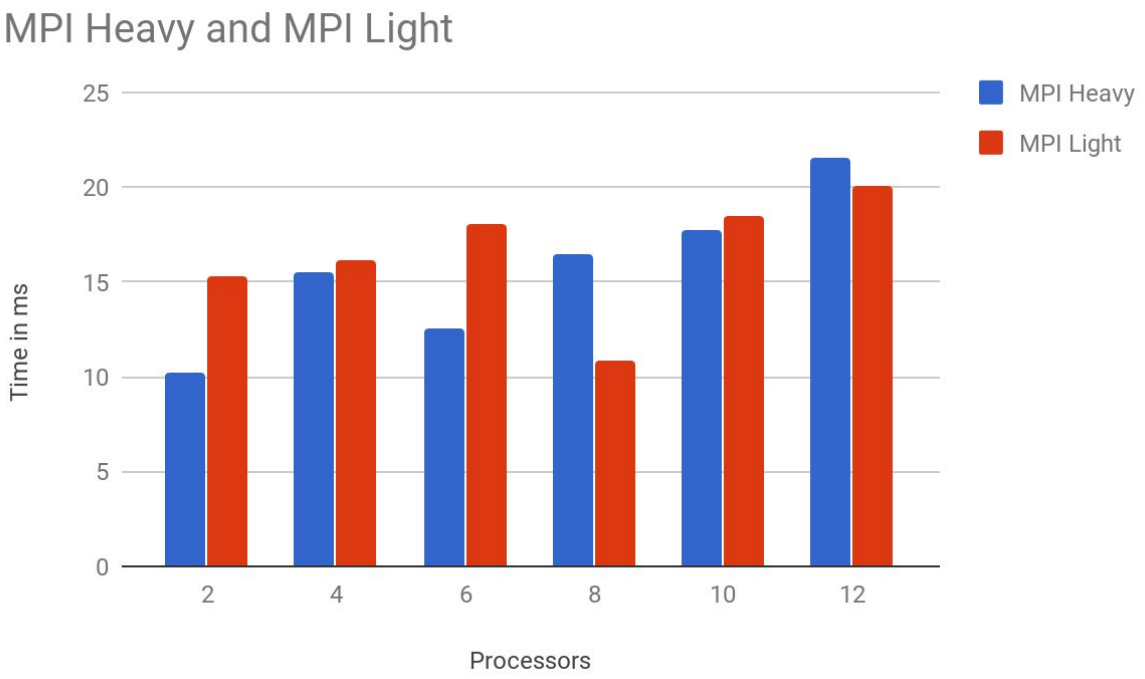


**Figure 4: Comparison of Heavily and Lightly Loaded processors wait-time for the MPI barrier part of the mixed OpenMP-MPI barrier**

**Analysis of experiments Conducted :**

There were many interesting observations that were made from the experiment that we have conducted. From the above graphs, Figure 1 illustrates the wait-times of the threads in the OpenMP barriers that were implemented. It is clear that under heavy load MCS barrier performs better than Dissemination barrier. This can be accounted to the message passing overhead in the dissemination barrier when the faster threads keep waiting for the slower ones and after the slower threads enter the barrier a lot of messages are exchanged for the remaining threads to know the completion of the slow ones. In the lightly loaded threads, this overhead does not seem to be a problem because all the threads finish the jobs relatively quickly and there is no delay in exchanging messages.

Figure 2 shows the wait-times for processors on the MPI Barriers under heavily loaded and lightly loaded conditions. Although both the barriers show similar performance, dissemination barrier seems to perform better for lower number of processors. Overall tournament barrier seems to scale better because of the log n number of message passes as opposed to n log n number of message passes in dissemination barrier.

From figures 3 and 4, we observed that irrespective of the load on the threads the wait times are the same for the number of threads or processors for both the barriers. One possible reason for this would be because all the threads of a processor run on the same

processor and not letting us exploit true concurrency of the cluster. This must have resulted in the skewed results. This also explains why the OMP threads had their wait times increasing at a very high rate.

**Work Division :**

The dissemination barrier of OpenMP and Tournament Barrier of MPI were implemented by Jyoti and the MCS tree barrier of OpenMP & dissemination barrier of MPI were implemented by Raj. We distributed the workload in a way that each one of us will gain knowledge of both OpenMP and MPI. Hence we implemented one barrier each for OpenMP and MPI. Combined barrier OpenMP-MPI was implemented by Raj . After integrating the tests harness was implemented by Jyoti.

The tests and experiments were done by both on all barriers to learn about pattern and efficiency of each. The documentation part was evenly divided where Jyoti did plotting and noting of graph values and Raj did other half. Documentation part was also equally divided.